ProSense

Cal Poly CPE Senior Project
Advisor: John Seng
June 2016

Johnny Favazza
Casey Glasgow
Matt Epperson

Table of Contents

Introduction

In conventional sports, athletes aim to outperform their opponents by utilizing coaches, analyzing film, and learning new techniques; a never ending evolution of skill. In more recent years, the introduction and integration of technology has proven crucial to advancing the performance of professional teams within the NFL, U.S. Olympic Teams, and the increasingly popular board sports. While this infusion of sciences into sports originated in the professional realm, technologies have trickled into the lives of everyday citizens in applications such as portable STIM machines, high school football helmets outfitted with accelerometers to sense concussions, and applications that help manage caloric intake. However, it is not just conventional athletes that seek to improve themselves. Board sports such as surfing, snowboarding, and skating all require fine tuning in order to improve. These board sports often do not use the traditional methods for improvement such as film or coaches and rely solely on practice.

This project aims to gather advanced data sets from MEMS sensors and GPS and deliver it to the user, who can capitalize on the data. The once *negligible* half-degree difference of your board barreling down a wave can be recorded from a gyro and exploited for the perfect turn. The *exact* speed dreaded by longboarders where speed wobbles turn into a road rash can be analysed and consequently avoided. Ascertaining the *summit* of your flight using combined GPS sensors from the ski ramp allows for the correct timing of tricks. When it comes to pursuing excellence in professional sports, amateur athletics, or a passionate hobby this project aims to advance personal performance and achievements.

Problem Statement

This goal of this project is to create a small printed circuit board that incorporates a variety of sensors in order to capture a rigid object's position, motion and orientation. The PCB will be enclosed in a waterproof case in order to allow the board to record data in most environments. The purpose for this device is to allow a user to take the captured data and play it back in a 3D simulation engine as a visualization tool for the user. This will be done by transmitting the data from the PCB to a computer or smart phone over USB or Bluetooth. The application will then perform post processing on the data and render it as a 3D rigid object in a simulated world where the motion can be played back by the user. We would also like to include a satellite view with gps markers overlaid on top. This dual visualization will give the user an idea of not only the fine grain details of motion through space but global knowledge with landmarks for reference and distance traveled across terrain. The data gathered will allow for the analysis and display of many statistics, including distance travel, airtime, or degrees of rotation during a spin.

Firmware

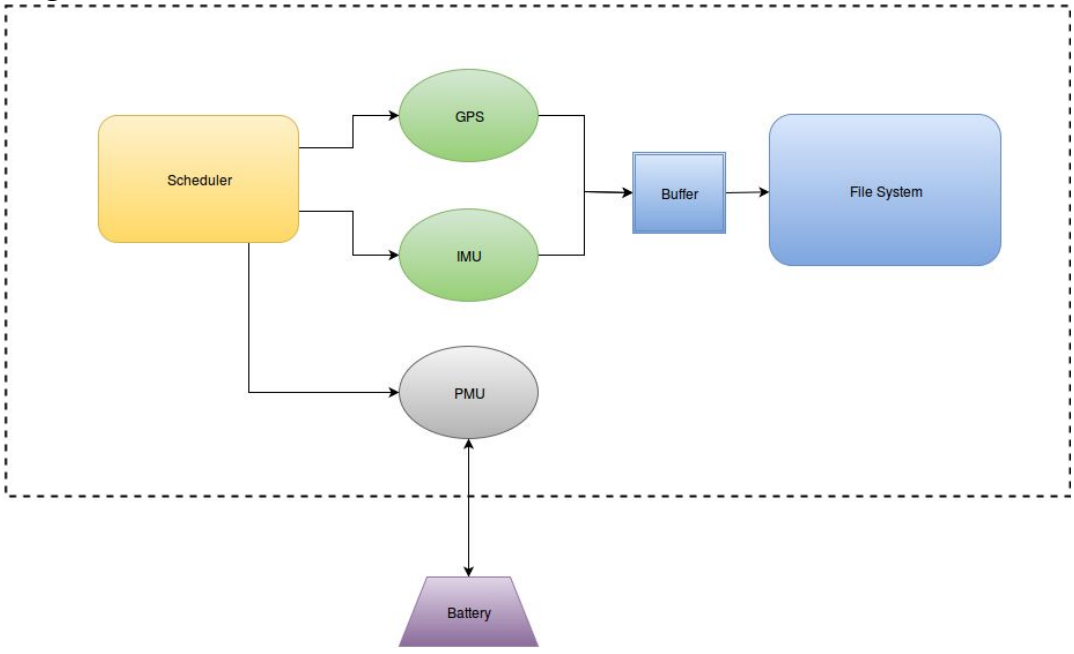
Overview

Our system requires us to record sensor data at specific rates. That data is then maintained in a file system. This imposes real time constraints which must be met in order for the the system to work successfully. In order to meet these constraints, we chose a powerful microcontroller that has hardware/interrupt support for most kinds of serial protocols that we need to communicate with peripherals. The three main parts of the firmware are device drivers, a scheduler, and a file system. Each peripheral (IMU, GPS, and memory) required a complete device driver to be built upon one of the hardware supported serial communication protocols such as UART, I2C, or SPI.

Our code was compiled via the arm-none-eabi-gcc compiler and uploaded via a rom bootloader that is accessed by pulling a RESET pin low followed by pulling the ISP pin low.

Big thanks to MicroBuildr[1] for providing a codebase along with tutorials for getting started with development for the LPC1114 line of microcontrollers.

Block Diagram



1. Code base found at: https://github.com/microbuilder/LPC1114CodeBase

Device Drivers
Each device driver had code for initializing and configuring the device and reading and writing data to the device. The device drivers were derived from each product's data sheet.

Scheduler
The scheduler's main purpose is to query sensors for data at specific intervals (Table 1). The scheduler is driven by the system tick timer which generates an interrupt every 1 ms. Tasks are handled by keeping a FIFO of task function pointers. There is no preemption to guarantee that a task will finish within in a certain time; however, each task was carefully considered and timed in order to make sure each task would finish in time for the next to be run.

Device	Part Name	Update Rate	Type of Data	Data Format	Size (bytes)
Inertial Measure-ment Unit	BNO055	100 Hz	Quaternion, Linear Acceleration	Quat = {u16 x, u16 y, u16 z, u16 w} Laccel = {u16 x, u16 y, u16 z}	14
Global Positioning System	MTK3339-PA6H	10 Hz	Time, Date, Latitude, Longitude, Altitude, Velocity, Heading	GPS_t= {float time, u32 date, float lat, float lon, float alt, float velocity, float heading}	28
Flash Memory	W25QXX	6.56 Hz	Page	Page_t = { u8 buffer[256] }	256

Table 1: List of Data types used in ulogfs

File System
The filesystem is loosely based off a log file system where each file is written sequentially and therefore called ulogfs (Micro Log File System). It's important to note that this filesystem is very bare bones as our users do not need 90% of the features of most filesystems. What the file system really lets the user do is record data for a given amount of time called a "run" and have that data be deliminated from other runs in files. These file when downloaded to a host PC can then be played back individually.

A file consists of an inode that stores metadata for that file such as time written, size, and a pointer to the start of the next file. We choose NAND flash memory for a combination of its price and memory density, however, NAND flash has a few properties that make it difficult to work with. Data must be written in pages of 256 bytes and can only be erased in blocks of 4 kbytes, 16 kbytes and 32 kbytes. In order to delete data in only a single file we kept files aligned to 4 kbytes.

Software

Overview
The software run on the host PC is split between two python scripts for post processing the data dumps from the device and a Unity application for visualizing data.

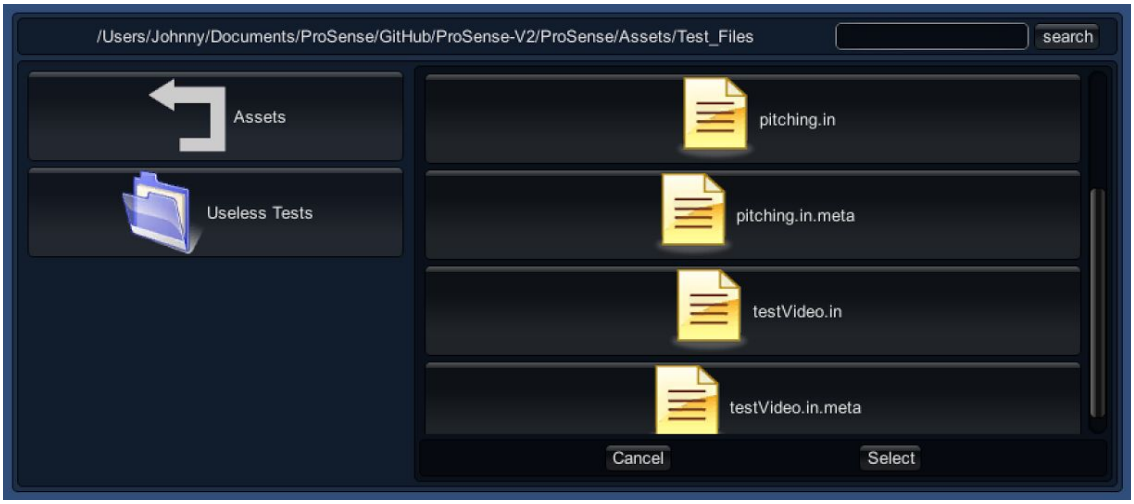
Modules

Post Processing
Our hardware setup allows us to interface with the host PC via a serial connection both for firmware updates and dumping the file system as hex file. That hex file needs to be processed into something that the can then be read into Unity. In order to do this, we wrote python scripts that convert the hex file into a human readable ascii file that Unity can use to read data from for visualization processing.

Unity
In order to visualize the data collected from the sport session, we used Unity to develop a desktop application in C# that shows the user their exact movements down to a fraction of a degree. The goal of this application is to provide precise feedback for those aspiring to increase their skills in their sport, fun highlights of the day for casual athletes, or friendly competition amongst friends to see who reached the top speed or landed the coolest trick.

The ProSense application starts the user at a file browser prompting them to choose which data file they want to visualize. This file browser displays recorded data of the day and directories to other stored data. After selecting a file, the user will be taken to the visualization screen where they can rewatch their run and analyze their movements more precisely than using traditional camera footage. This visualization process was created using Unity's built in rotation and transformation functions. After the user finishes watching the visualization, they will be taken back to the file browser to select a new file to watch or exit the application.

File Browser
The file browser was made using a free asset created by user *CombustableJo* downloaded from the Unity Store. The asset came with all of the images used to display the prompt, files, and directories. It allowed for some customizable options such as displaying the search bar, searching recursively, or various color options. We used the provided public methods in the FileBrowser class to draw the window and select the file path. Below is an in-application screenshot of the file browser screen.



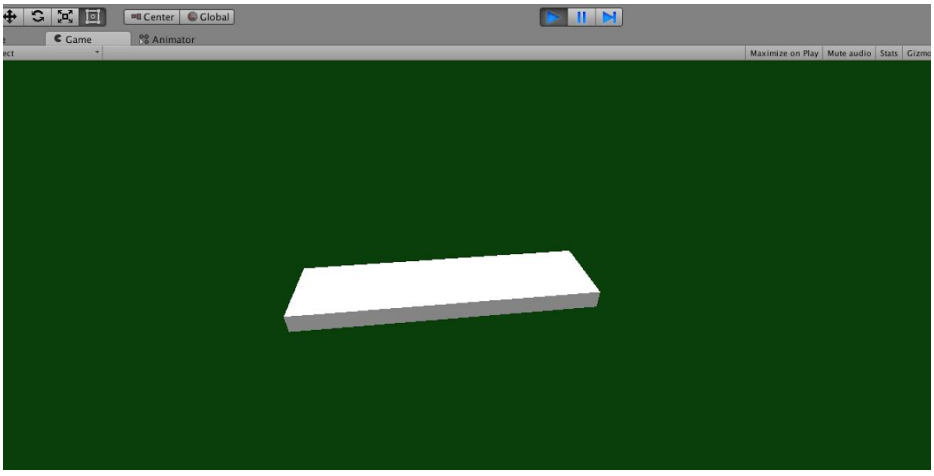
Screenshot of the file browser displaying various recorded data used for testing.

Visualization

The visualization process was created using Unity's built in rotation and transformation functions and it's animation process. At the start of the visualization, a new Unity scene is created that initializes the rectangular block (will substitute a surfboard or snowboard depending on the chosen activity), the background colors, and buffers the selected data file.

The data file gets buffered into a new file of roll, pitch, and yaw to be used for the rotating of the object. We created a function called BufferRotation(string path) that takes in a file and opens it. It then reads from this file and writes the roll, pitch, and yaw to a new file separated by a new line. After this finishes, the visualization of the run can begin.

To show the object moving, we used Unity's Rotate() function which takes roll, pitch, and yaw as parameters. Rotate() is called in Unity's Update() function, which is called once per frame. The rectangular block is tied to Unity's animation editor which allows for smooth transitions between states. When the simulation has finished, it return the user to the file browser so they can select another file or quit the application.



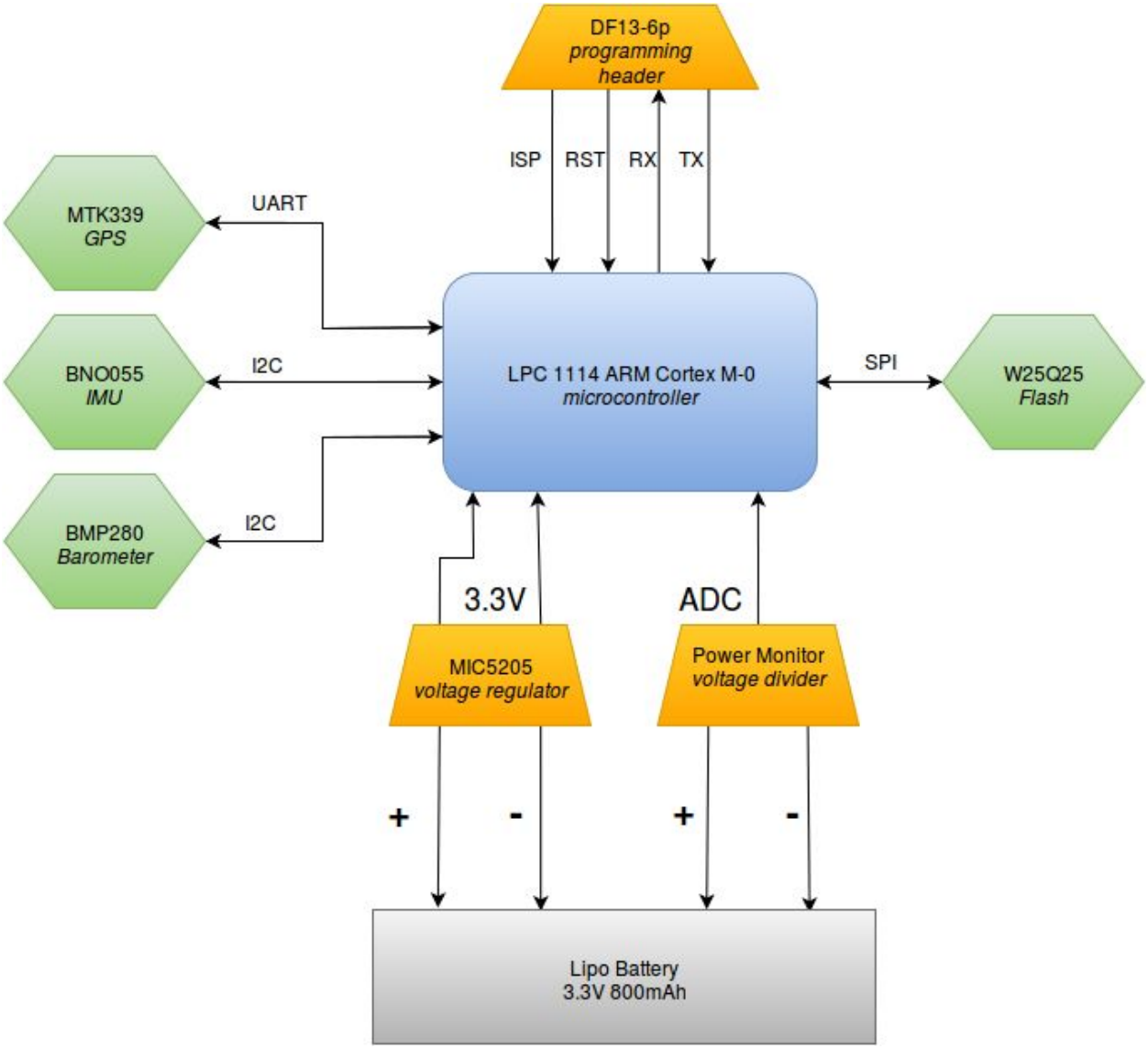
Screenshot of a visualization of a rectangular block replaying movements captured by the ProSense board

Hardware

Overview

The goal for our hardware development was to find a balance between creating the smallest form factor with the longest battery life. Our design consists of a single 3.3 volt microcontroller to interface and control each peripheral. The peripherals can be broken into sensors, memory, or power management.

Block Diagram



Modules

Microcontroller

Our first hardware iteration used an ATMega328p as the microcontroller because it was simple to upload firmware to and had lots of community support. We quickly realized that it would not be powerful enough nor did it have enough hardware I/O peripherals for connecting to all our devices. It did let us quickly prototype the idea and test several components individually like the GPS. We next moved to the LPC800 which is one of the only breadboard friendly ARM Cortex M-0 microcontrollers on the market which made it very easy to prototype. The main problem we ran into with this microcontroller was that it did not run at fast enough speeds and was very limited in program memory (4kB) and SRAM (1kB). We moved onto its older brother the LPC1114 which runs at 50Mhz and has a 32kB flash memory and 4kB SRAM which is more than enough for our project. To date we have designed and manufactured three different custom PCB's using this microcontroller. The current package type we are using is QFP-33 and is readily available from most electronics providers.

The LPC1114 has hardware support for UART, I2C and SPI. This was very important for us since being able to run these peripherals off interrupts off loaded a significant amount of work. It also has a several ADC and quite a few GPIO pins that we used for monitoring battery voltage as well as flashing LEDs.

Sensors

For the IMU we chose to use the Bosch BNO055 which is a specialized 9DOF IMU that provides sensor information that has already been passed through an Extended Kalman Filter (EKF) and is outputted in the form of either euler angles, quaternions, or linear acceleration at 100 Hz. The benefit of having the data already filtered means that we did not have design and tune an EKF ourselves which would have been a significant project in itself. In addition, having the information outputted in quaternions is ideal for visualizing data in 3D as it avoids the unity issue associated with euler angles. The downside to this chip was its footprint which was very non-standard and had a small pitch that made manufacturing by hand very difficult. Our success rate for soldering this part is around 50%.

The GPS module we decided upon is the MTK3339 by GlobalTop. It receives geospatial information at maximum update rate of 10 Hz which is then transmitted to the microcontroller over UART. The real deciding feature of the MTK3339 was its builtin in ceramic patch antenna that removed the need for us design the RF components. The chipset has an easy to solder package that makes manufacturing simple. The downsides to this chip is that it uses UART to communicate. Normally this wouldn't be a problem; however, it is a major obstacle for us since we upload firmware via UART causing a bus contention. In the end we added a jumper to disconnect GPS module during firmware updates which is not ideal but livable.

Memory

Choosing a flash memory chip was difficult since the range of options was fairly wide. We ended up choosing the Winbond W25Qxx series NAND flash memory for its high communication speeds over SPI, memory size and low power usage. The current chip in the

series we are using is the W25Q125FV which has a memory capacity of 16MB, can be run up to 104 Mhz and comes in the WSON 8x6mm package.

Power

Our design requires the embedded system be able to be continuously run for several hours at a time. Lithium Polymer (LiPo) batteries are a common choice for such application because of their high energy density. The nominal operating voltage of a LiPo is 3.7 - 4.4V so we therefore needed to step this voltage down to 3.3V to be compatible with our microcontroller and sensors. In order to do this we chose the MIC5205-3.3v for its size and low voltage dropout characteristics.

One downside of LiPo's is that they require special care in order to prevent damaging them. Discharging a LiPo cell past 2.7V will cause it to malfunction and no longer be able to hold a charge. So in addition to the voltage regulator we have a single ADC pin of our microcontroller connected to a voltage divider that is connected to the power supply in order to monitor battery voltage. This allows us to automatically enter a low power mode as well as alerting a user that the battery is running low through an LED.

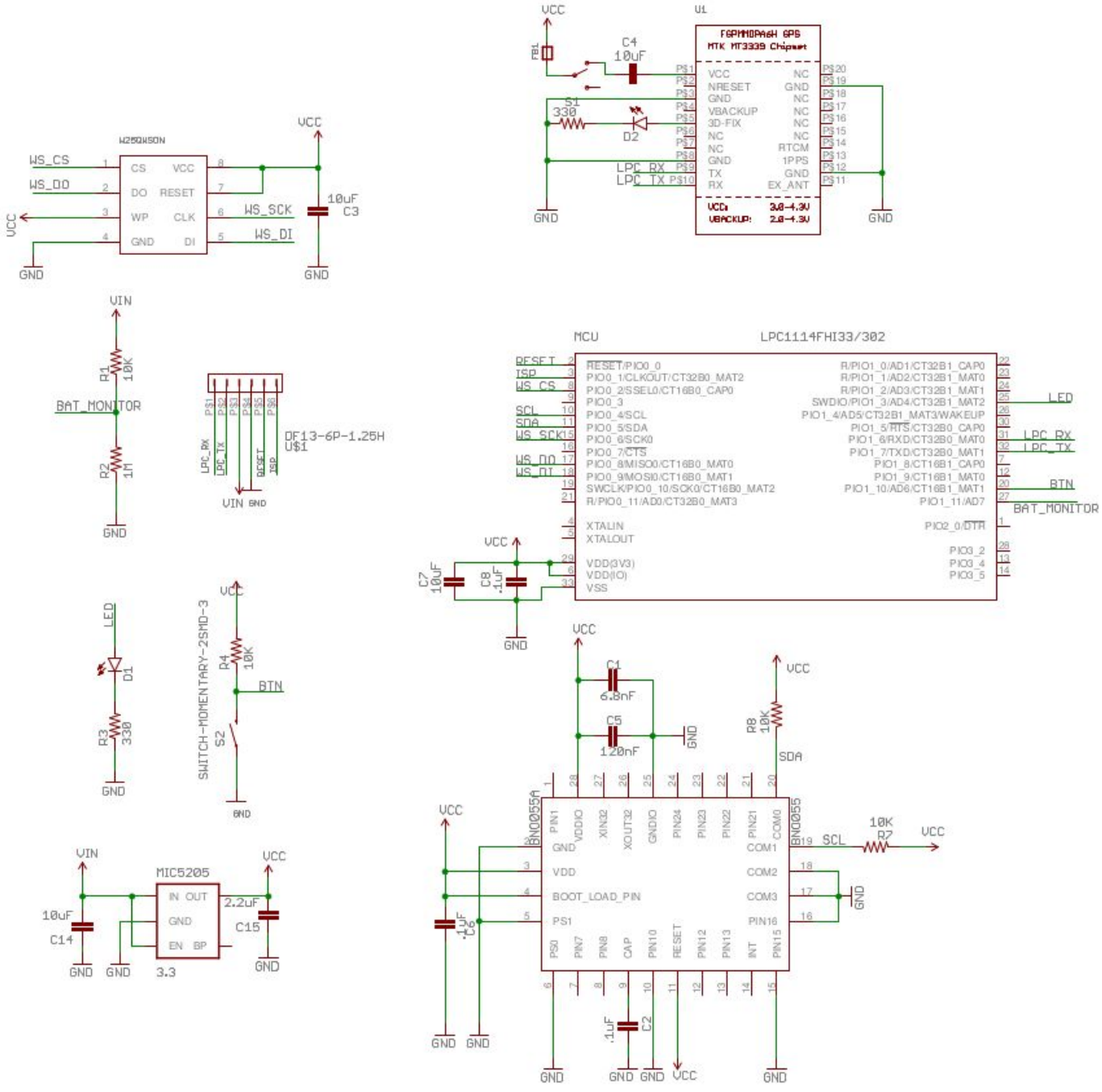
Power Consumption

Module	Device	Average Power	Max Power
GPS	MTK3339	66mW	82mW
MCU	LPC1114	16.5mW	23.1mW
IMU	BNO055	40.6mW	40.6mW
Memory	W25Q125FV	20mW	25mW
LED	LT Q39G-Q1S2-25-1-5	5mW	5mW

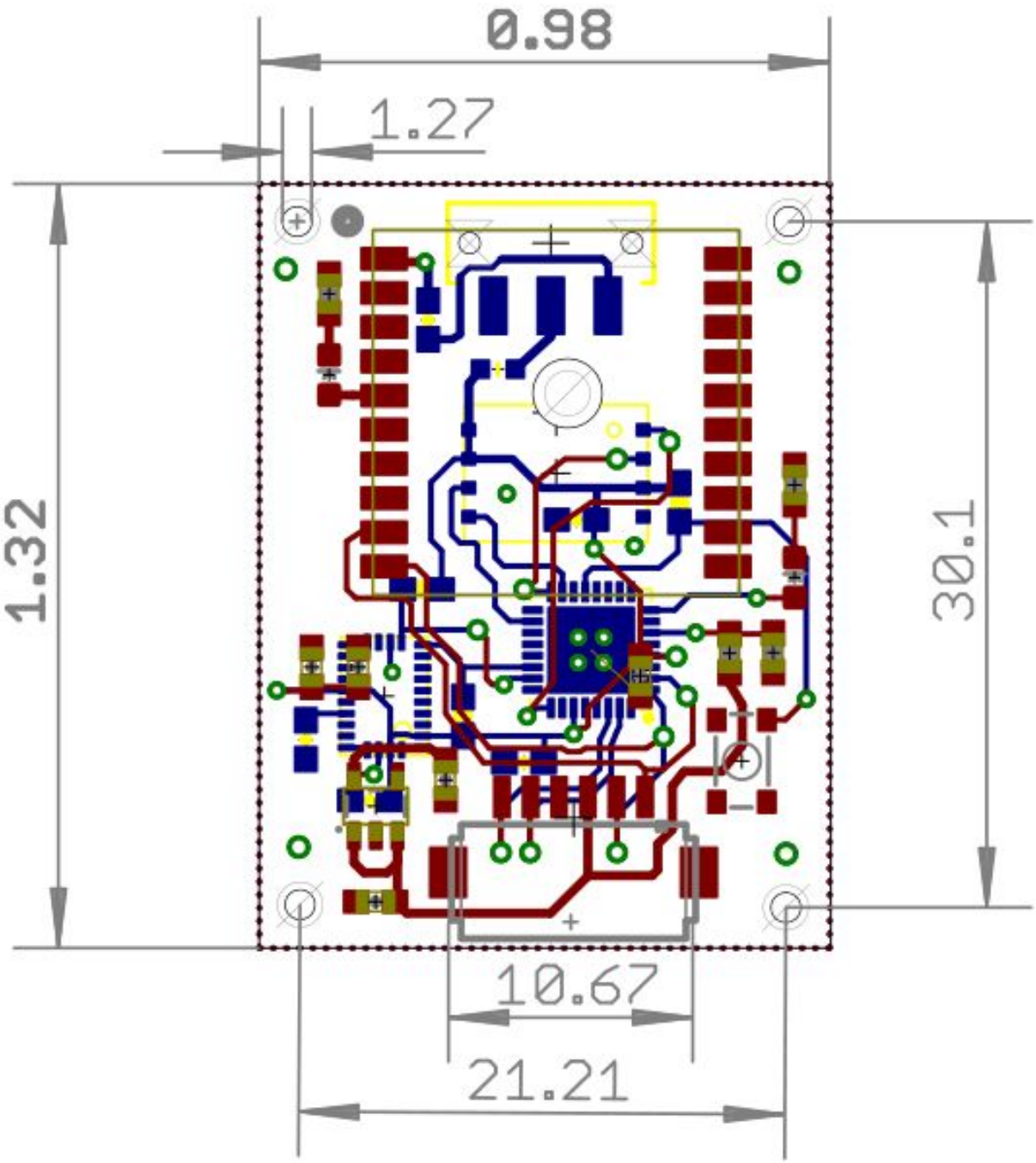
Table 2: Power consumption of major components

Run Time = mAh * Voltage / power consumption = 400mAh * 3.3V / 148.1mW = 8.913 Hours

Electrical


W25Q16SON
WS_CS
WS_DO
WS_SCK
WS_DI
VCC
GND
10uF
C3
R1
10K
BAT_MONITOR
R2
1M
VIN
DF13-6P-1.25H
U$1
LED
D1
R3
330
R4
10K
BTN
S2
SWITCH-MOMENTARY-2SMD-3
MIC5205
IN OUT
GND
EN BP
2.2uF
C15
C14
3.3
U1
FGPMMOPA6H GPS
MTK MT3339 Chipset
C4
10uF
S1
330
D2
LPC_RX
LPC_TX
VCC: 3.0~4.3V
VBACKUP: 2.0~4.3V
MCU
LPC1114FHI33/302
RESET
ISP
WS_CS
SCL
SDA
WS_SCK
WS_DO
WS_DI
LED
BTN
BAT_MONITOR
XTALIN
XTALOUT
VDD(3V3)
VDD(IO)
VSS
C7
10uF
C8
.1uF
C1
6.8nF
C5
120nF
R8
10K
SDA
SCL
R7
10K
BNO055
BNO055A
C6
.1uF
C2
.1uF

Mechanical

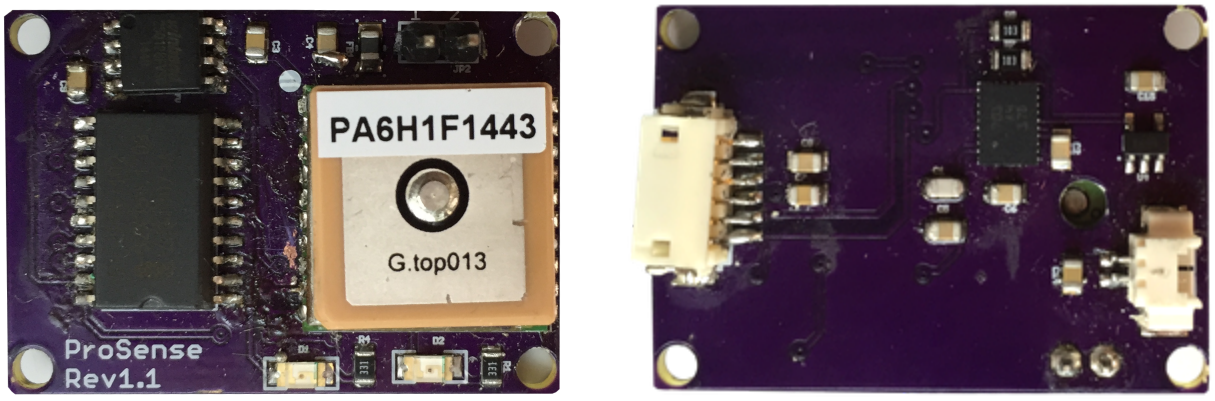


Manufacturing

One of the major learning curves of this project was PCB assembly. In the first PCB we created we designed the board to use very small footprint SMD for passive components which turned out to be beyond our skill at the time to hand solder. This was a valuable lesson for the next iteration of boards where we learned to only choose components that were hand solderable. This worked well in the beginning but as we tried to make the boards smaller and smaller with higher grade components we needed new methods of manufacturing our PCBs. Eventually after much research into DIY PCB assembly we purchased a hot air rework station, along with solder paste, and stencils. This combination proved to be very effective and we can now solder down to 0402 for passives and even SMD components without leads and very small pitch.

The method for soldering with the hot air station is as follows. First place PCB on flat surface and attach stencil to the top. Next apply solder paste to the top and use credit card to spread evenly over entire surface. Very carefully to not smear any solder paste applied to surface of PCB remove the stencil. At this point you should have an even coat of solder paste on all copper surfaces and can begin placing SMD components. For this part we used a magnifying glass and tweezers to pick and place the components by hand. After all components have been placed fire up the hot air rework station. Its best to consult each product data sheet for information on reflow characteristics. A good rule of thumb is to spend about a minute with the hot air 2-3 inches above the PCB to allow it to come gently up to temperature and then move in closes to reflow the solder.

Our boards were ordered through OSHPark which provides decent quality boards at very inexpensive prices. Stencils were purchased through OSHStencil for reasonable prices as well.



ProSense version 1.1 completely soldered and ready for testing

Bill of Materials

Original Cost Estimate
From the beginning of the project, we estimated the cost of the development to be relatively cheap. Small ARM microcontrollers only cost a few dollars each, and the additional components required to get it up and running only cost a few dollars more per board. The large bulk of the cost will come from the peripherals connected to the MCU. The major peripherals will be the flash memory, IMU, and GPS. We estimated the cost of the memory to be around $5-$10, the IMU will be around $10-$15, and the GPS will be around $25-$35. There will also be additional costs for soldering tools, manufacturing the PCBs, and possibly developing a case for the device. We estimated the total cost of components for one device to be somewhere in the range of $50 to $100, depending on the type and quality of the components used.

Throughout development, we tested various components and eventually narrowed the choices down to our current list. The following table lists all the components used in the current design and their cost.

Part	Value	Device	Description	Cost ($)
BNO055	------	BNO055A	Inertial Measurement Unit Absolute Orientation 9-Axis	10.99
C1	6.8nF	VJ0603Y682KXQPW1BC	Multilayer Ceramic Capacitor (MLCC) - 10 volts X7R 10%	0.10
C2	0.1uF	0603ZC104KAT2A	MLCC - 10 volts X7R 10%	0.28
C3	10uF	GRM188R61A106KE69D	MLCC - 10 volts X5R 10%	0.08
C4	10uF	GRM188R61A106KE69D	MLCC - 10 volts X5R 10%	0.08
C5	120nF	0603ZC124KAT2A	MLCC - 10 volts X7R 10%	0.28
C6	0.1uF	0603ZC104KAT2A	MLCC - 10 volts X7R 10%	0.28
C7	10uF	GRM188R61A106KE69D	MLCC - 10 volts X5R 10%	0.08
C8	0.1uF	0603ZC104KAT2A	MLCC - 10 volts X7R 10%	0.28
C14	10uF	GRM188R61A106KE69D	MLCC - 10 volts X5R 10%	0.08
C15	2.2uF	GRM188R61A225KE34D	MLCC - 10 volts X5R 10%	0.067
D1	Green	LT Q39G-Q1S2-25-1-5	LED	0.146
D2	Red	LS Q976-NR-1	LED	0.067

FB1	-----	BLM18EG601SN1D	Ferrite Bead	0.126
MCU	-----	LPC1114FHI33/302	32-bit ARM Cortex-M0 microcontroller	3.07
MIC5205	-----	MIC52053.3V	150mA Low-Noise LDO Regulator	0.62
R0(?)	330 Ω	CRCW0603330RFKEA	Thick Film Resistor - 1% 1/10WATT	0.044
R1	10K Ω	CR0603-JW-103ELF	Thick Film Resistor - 5% 1/10WATT	0.016
R2	1M Ω	CR0603-JW-105ELF	Thick Film Resistor - 5% 1/10WATT	0.016
R3	330 Ω	CRCW0603330RFKEA	Thick Film Resistor - 1% 1/10WATT	0.044
R4	10K Ω	CR0603-JW-103ELF	Thick Film Resistor - 5% 1/10WATT	0.016
R7	10K Ω	CR0603-JW-103ELF	Thick Film Resistor - 5% 1/10WATT	0.016
R8	10K Ω	CR0603-JW-103ELF	Thick Film Resistor - 5% 1/10WATT	0.016
S1	-----	JS102011SAQN	Slide Switches 1PDT .3A SMT	0.51
S2	-----	CKN10502CT-ND	SWITCH TACTILE SPST-NO 0.05A 16V	0.28
U1	-----	GPS_FGPMMOPA6H	GPS Module - MTK MT3339 Chipset	29.95
U$1(?)	-----	DF13-6P-1.25H(50)	Headers & Wire Housings 1.25MM RA PIN HEADER 6P SMT GOLD	1.18
W25Q	-----	W25Q128FVWSON	W25Q128FV - SPI Flash Memory	1.725
			TOTAL	**50.439**

Table 3: Bill of Materials for PCB Layout

The total cost of components for one board comes out to just over $50. This is at the low end of our original estimate, but does not include the cost of the PCBs, soldering supplies and tools, or the battery. The PCBs only cost $5 for 3 boards, so the additional cost is minimal, and the cost of solder is negligible overall.

Lessons Learned

One important takeaway from this project is to make incremental progressions. For example we started with an 8 bit Atmel microcontroller and got our feet wet writing a device driver for the GPS unit. This got us started developing on microcontrollers and the process of setting up a development environment while also giving us a good start on a piece of code for working with the GPS. From there we moved up to a 32 bit ARM microcontroller that was much more difficult to work with. If we had just tried to jump into the project using the ARM chip we would have been discouraged quickly and wouldn't have made it far without the proper knowledge. This same lesson also applies to PCB manufacturing. Soldering SMD components can be very difficult and frustrating to debug. We started out using devices that had very easy packages to solder and moved to more difficult ones as our skills improved.

If we were to go back and redo this project one of the main things we would have done differently is how we managed our code development. First we would have adopted test driven development from the start as doing this in the beginning means that you build up unit tests for each part of the firmware that can be continuously tested and provides a consistent method of documentation since each unit test describes how that piece of code should work. There was a few times that new code we introduced broke existing code which wasn't caught until later on. Additionally we could have done a better job with source control. We rarely would do commits and when we did they ended up being very large which would enivitable end up messing with someone else's development.

Quick Notes

1. Debug hardware and firmware separately.
2. Use git more often and as a documenting tool
3. ALWAYS check for cold solder joint
4. Solder paste + Stencils + Reflow = Quicker, more reliable manufacturing
5. TDD/Unit Testing for embedded firmware is extremely useful.
6. Always buy extra caps/resistors
7. Try to source components from one vendor if possible
8. Breadboard and test individual device/device drivers separately and NOT on custom pcb.

Conclusion

This was an interesting project that took us through every aspect of creating an embedded device. It required us to come up with an initial concept design that, while ambitious enough in scope, still allowed us to make incremental progress towards individual goals. Throughout the process we learned not only technical skills such as taking a data sheet and writing a device driver from it or debugging race conditions, but also the logistics of embedded engineering like sourcing components and managing a budget.

Over the course of the project, we started from simple data collection using breadboarded components to a completely custom embedded device that is able to combine sensor information into powerful data sets that have the potential to aid the advancement of a user's ability for board sports.

Appendix

Links

https://github.com/matte1/ProSense-Eagle
https://github.com/matte1/ProSense
https://github.com/jfavazza/ProSense-V2.git

Code Sample

```
/*****************************************************************************/
/*!
        @file   ulogfs.c
        @author    Matt Epperson

        @section Overview

        This is a very primitive file system for storing information into flash
        memory. The idea is to have files be sector aligned because the minimum
        erase size is for a sector. There can be a supernode at the beginning
        that contains pointers to all the other files.
*/
/*****************************************************************************/

#include "ulogfs.h"

static ULOGFS uLogFS;


/*****************************************************************************/
/*!
    @brief Checks a page of data to see if its empty or has data in it.

    @return bool
        True if page is empty
*/
/*****************************************************************************/
static bool _ulogCheckPage(uint8_t buf[])
{
   int ndx;

   for (ndx = 0; ndx < 256; ndx++)
        if (buf[ndx] != 0xFF)
        break;

   return ndx == 256;
}

/*****************************************************************************/
/*!
    @brief This assumes a that the next free block is unkown and scans from
   beginning of memory trying to find the next free sector.
```

```
    @return address
        The address of the next avaliable block
*/
/*****************************************************************************/
static uint32_t _ulogNextFreeSector()
{
   // int ndx = 0;
   uint32_t address = 0;
   uint8_t buf[BLOCKSIZE];

   // The second conditional is actually unnecassary
   while (address < CFG_W25QXX_MAX_ADDRESS && address != 0xFFFFFFFF)
   {
       // Get pointer to the next Inode
       w25qReadPage(buf, address);
       uLogFS.lastInode = address;
       address = (buf[13] << 24) |
              (buf[14] << 16) |
              (buf[15] << 8)  |
              buf[16];
   }

   address = uLogFS.lastInode;
   while (address < CFG_W25QXX_MAX_ADDRESS)
   {
       w25qReadPage(buf, address);

       if (_ulogCheckPage(buf))
       return address;

       address += SECTORSIZE;
   }

   // TODO: If it gets to here that means address is > max address. Should
   // return an error really.
   return address;
}


/*****************************************************************************/
/*!
    @brief This will be called during system init in order to perpare for
   logging data. It will set a few variable such as curBlock and numBlocks

   TODO: 1) Add return codes for error handling. This could just return true
       if there is room avaliable and false if not

   @return 1
*/
/*****************************************************************************/
uint8_t ulogInit()
{
   uLogFS.lastInode = 0;
   uLogFS.bufferIndex = 0;
    uLogFS.curBlock = _ulogNextFreeSector();

   #ifdef CFG_ULOGFS_DEBUG
       printf("Init - CurBlock: %u%s", (unsigned int)uLogFS.curBlock,
       CFG_PRINTF_NEWLINE);
   #endif
```

```
    return 1;
}

/*****************************************************************************/
/*!
    @brief Writes buffer 'buffer' of size 'size', which represents PART of a file's
    content to the file system.

    NOTE: Need to flush data on the very last BufferData that is called

    @param[in] data
        Data to copy to the internal buffer

    @param[in] length
        Length of data to copy to the internal buffer
*/
/*****************************************************************************/
void ulogBufferData(uint8_t *data, int length)
{
    int spaceLeft = BLOCKSIZE - uLogFS.bufferIndex;

    // If we have enough room to copy all the data into the buffer do so
    // Otherwise copy what we can and write it to flash
    if (spaceLeft >= length)
    {
        memcpy(&uLogFS.buffer[uLogFS.bufferIndex], data, length);
        uLogFS.bufferIndex += length;
    }
    else
    {
        memcpy(&uLogFS.buffer[uLogFS.bufferIndex], data, spaceLeft);
        w25qWritePage(uLogFS.buffer, uLogFS.curBlock, BLOCKSIZE);
        memcpy(uLogFS.buffer, &data[spaceLeft], length - spaceLeft);
        uLogFS.bufferIndex = length - spaceLeft;
        uLogFS.curBlock += BLOCKSIZE;
    }
}

/*****************************************************************************/
/*!
     @brief Creates a new file by writting a new Inode. There is a lot of unused
    space inside these Inodes that should be used to track file stats. Number
    of sectors used, time ended, last Inode for faster file searching.
*/
/*****************************************************************************/
void ulogNewFile()
{
     uint8_t buf[13];
    uint32_t systime;

    systime = systickGetMillisecondsActive();

    buf[0] = BLOCK_TYPE_INODE;

    // Write Start Time
     buf[1] = (systime >> 24) & 0xFF;
    buf[2] = (systime >> 16) & 0xFF;
    buf[3] = (systime >> 8) & 0xFF;
    buf[4] =  systime & 0xFF;
```

```
    // TODO: Add GPS Time, Date if possible
    // if (gps_available)
    // {
    buf[5] = '1';
    buf[6] = '2';
    buf[7] = '3';
    buf[8] = '4';
    buf[9] = '5';
    buf[10] = '6';
    buf[11] = '7';
    buf[12] = '8';
    // }

    w25qWritePage(buf, uLogFS.curBlock, 13);

    // Update curBlock
    uLogFS.curBlock += BLOCKSIZE;
}

/*************************************************************************/
/*!
    @brief Cleans up after finished writing a file. This means flushing any
    data in the buffer. NOTE It also writes the next address back to the previous
    inode even if that new inode has not been created yet. Were making the
    assumption that it will be created
*/
/*************************************************************************/
void ulogCloseFile()
{
    uint32_t fileSize = uLogFS.curBlock - uLogFS.lastInode - BLOCKSIZE + uLogFS.bufferIndex;
    uint8_t buf[30];
    memset(buf, 0xFF, 30);

    // Flush any data in buffer
    if (uLogFS.bufferIndex > 0)
    {
         w25qWritePage(uLogFS.buffer, uLogFS.curBlock, uLogFS.bufferIndex);
        uLogFS.curBlock += BLOCKSIZE;
        uLogFS.bufferIndex = 0;
    }

    // Make sure we are sector aligned for new file
    if (uLogFS.curBlock % SECTORSIZE != 0)
        uLogFS.curBlock = ((int)(uLogFS.curBlock / SECTORSIZE) + 1) * SECTORSIZE;

    // Setup data to write to last Inode
     buf[13] = (uLogFS.curBlock >> 24) & 0xFF;
    buf[14] = (uLogFS.curBlock >> 16) & 0xFF;
    buf[15] = (uLogFS.curBlock >> 8)  & 0xFF;
    buf[16] =  uLogFS.curBlock & 0xFF;

    buf[17] = (fileSize >> 24) & 0xFF;
    buf[18] = (fileSize >> 16) & 0xFF;
    buf[19] = (fileSize >> 8)  & 0xFF;
    buf[20] =  fileSize & 0xFF;

    // NOTE: This wont overwrite the first 13 bytes since unless some
    // of those bytes are 0xFF. But since we memset the array to 0xFF
    // anyways we should be good.
```

```
   w25qWritePage(buf, uLogFS.lastInode, 21);

   // Update where the last Inode was
   uLogFS.lastInode = uLogFS.curBlock;
}

/*****************************************************************************/
/*!
   @brief Goes through file system and prints out all files and the
       time that they were created.
*/
/*****************************************************************************/
void ulogListFiles()
{
   uint32_t fileTime, size, address = 0;
    uint8_t buf[BLOCKSIZE];

   w25qReadPage(buf, address);

   while (buf[0] != 0xFF && address < CFG_W25QXX_MAX_ADDRESS)
    {
        if (buf[0] == BLOCK_TYPE_INODE)
        {
                fileTime = (buf[1] << 24) |
                                    (buf[2] << 16) |
                                    (buf[3] << 8)  |
                                        buf[4];

       // Size
                size = (buf[17] << 24) |
                               (buf[18] << 16) |
                               (buf[19] << 8)  |
                                buf[20];

       printf("Found File at Time %u at %u with a size %d%s", (unsigned int)fileTime,
                       (unsigned int)address, (unsigned int)size, CFG_PRINTF_NEWLINE);

       // Next Address
                address = (buf[13] << 24) |
                                    (buf[14] << 16) |
                                    (buf[15] << 8)  |
                                     buf[16];

        }
       else
       {
       printf("No File at %u%s", (unsigned int)address, CFG_PRINTF_NEWLINE);
       return;
       }
       w25qReadPage(buf, address);
    }

}
/*****************************************************************************/
/*!
   @brief Prints out all blocks of data starting at the fist byte.
*/
/*****************************************************************************/
void ulogPrintFileSystem()
{
```

```
    uint32_t address = 0, temp;
    uint8_t buf[BLOCKSIZE];

    // The second conditional is actually unnecassary
    while (address < CFG_W25QXX_MAX_ADDRESS && address != 0xFFFFFFFF)
    {
        // Get pointer to the next Inode
        w25qReadPage(buf, address);
        temp = address;
        address = (buf[13] << 24) |
                  (buf[14] << 16) |
                  (buf[15] << 8)  |
                  buf[16];

        while (temp < address && !_ulogCheckPage(buf))
        {
        uartSend(buf, BLOCKSIZE);
        temp += BLOCKSIZE;
        w25qReadPage(buf, temp);
        }
    }
}
```